

# Combating Spam and Denial-of-Service Attacks with Trusted Puzzle Solvers<sup>\*</sup>

Patrick P. Tsang and Sean W. Smith

Department of Computer Science  
Dartmouth College  
Hanover, NH 03755, USA  
{patrick,sws}@cs.dartmouth.edu

**Abstract.** Cryptographic puzzles can be used to mitigate spam and denial-of-service (DoS) attacks, as well as to implement timed-release cryptography. However, existing crypto puzzles are impractical because: (1) solving them wastes computing resources and/or human time, (2) the time it takes to solve them can vary dramatically across computing platforms, and/or (3) applications become non-interoperable due to competition for resources when solving them.

We propose the use of *Trusted Computing* in constructing crypto puzzles. Our puzzle constructions have *none* of the drawbacks above and only require each client machine to be equipped with a small tamper-resistant *Trusted Puzzle Solver (TPS)*, which may be realized using the prevalent Trusted Platform Module (TPM) with minimal modifications.

## 1 Introduction

### 1.1 Cryptographic Puzzles

Cryptographic puzzles are problems that require a designated amount of time and/or resources to solve. Since 1978 when Merkle first proposed them for securing key agreement [23], crypto puzzles have been used to overcome a range of security challenges.

**Proof-of-Work Puzzles** Spammers try to send as many spam emails (i.e., unsolicited bulk emails) as possible to maximize their profits; attackers can take down a web server by requesting many webpages within a short period of time. Unfortunately, although it is well-known that charging fees on service accesses would provide the necessary disincentive for abuses as such, there is no practical way to charge money in the electronic world today. One major use of crypto puzzles is to impose costs on the clients by forcing them to do some work per

---

<sup>\*</sup> This work was supported in part by the U.S. Department of Homeland Security under Grant Award Number 2006-CS-001-000001, the Institute for Security Technology Studies, under Grant number 2005-DD-BX-1091 awarded by the Bureau of Justice Assistance, and the National Science Foundation, under grant CNS-0524695. The views and conclusions do not necessarily represent those of the sponsors.

service access (and hence the name “Proof-of-Work” [17]), thereby consuming their resources, e.g., CPU cycles.

Proof-of-Work puzzles are also known as *Client Puzzles* [2,18,31], especially when they are used to mitigate denial-of-service (DoS) attacks at the lower layers of the communication protocol stack such as the network layer and the transport layer. Client Puzzles have the additional property that their generation, as well as the verification of their solution must be done efficiently because otherwise these two operations would become new DoS attack surfaces.

There are scenarios when having the ability to obtain and solve Proof-of-Work puzzles before the actual service accesses is desirable. For example, allowing the puzzles to be “presolved” hides from the user the latency of solving the puzzles when trying to access web servers for webpages, and can still rate-limit accesses from the user. However, presolvable client puzzles are less effective in mitigating DoS attacks because the adversary can accumulate enough puzzle solutions and use all of them at the same time.

Dwork and Naor were the first to propose the use of crypto puzzles for fighting spam [13]. Back independently invented “hashcash” [3]. Other applications of Proof-of-Work puzzles include metering visits to websites [15], providing incentives in peer-to-peer systems [27], mitigating (distributed) DoS attacks [21,2,12], rate-limiting TCP connections [18] and defending against Sybil attacks [7]. Finally, Roman et al. [26] proposed a scheme that uses pre-challenges to fight spam, in which the pre-challenges can range from, e.g., security questions to micro-payments to CAPTCHAs [30]. While our work focuses on constructing better crypto puzzles, their work provides insights on several compatibility and usability issues when one deploys our solution on existing email infrastructures.

**Time-Lock Puzzles** May [22] first discussed the idea of sending information into the future, i.e., encrypting a message so that no one can decrypt it until a predetermined time. Rivest et al. [25] later formally proposed *Timed-Release Cryptography (TRC)* and *Time-Lock puzzles*—crypto puzzles that can be solved only after a predetermined time—and their use to realize TRC.

The algorithm for solving Time-Lock puzzles must be non-parallelizable, i.e., a Timed-Lock puzzle with multiple machines won’t be any faster than solving it with a single machine. For instance, puzzles that ask for the preimage of hash values are bad Time-Lock puzzles because computing hash preimages can be parallelized. Rivest et al.’s Time-Lock puzzles [25] ask for a series of modular squarings, the computation of which no one knows how to parallelize.

Applications of TRC and hence some Time-Lock puzzles include sealed-bid auctions [25], encapsulated key escrow [25,4], digital time capsule [24], and timed release of digital signatures [16] and commitments [6]. Chan and Blake proposed a timed-release encryption scheme that provides user anonymity [11] but requires a passive server. Some other timed-release constructions are [9,10].

## 1.2 Problems with Existing Crypto Puzzles

Existing crypto puzzles fail to effectively combat spams and DoS attacks, as they all suffer from one or more of the drawbacks below. As we will see, our design of crypto puzzles to be presented later in this paper has *none* of these drawbacks.

**Impreciseness** A problem faced by all existing crypto puzzles is that it is extremely difficult to precisely specify the time and/or resources required to solve a puzzle, mostly due to the heterogeneity of computing devices available today. For instance, solving a crypto puzzle that requires 1 minute to solve on a Desktop PC could take an hour on a PDA.

For Proof-of-Work puzzles, this impreciseness adds complication to, if not rendering it entirely infeasible, their use for defeating service abuse. For example, if one sets the difficulty of solving the puzzles assuming the presence of resourceful spammers, then legitimate users will probably be practically unable to send emails on their PDAs, or PCs they bought three years ago. The situation gets no better for Time-Lock puzzles: timed-release cryptography becomes insecure if Time-Lock puzzles can be solved faster than the puzzle creator expected. In Internet-based contests, for example, a more resourceful candidate can decrypt the test questions earlier, resulting in unfairness.

**Environmental and User Unfriendliness** Solving crypto puzzles consumes resources in the computing devices. Most existing constructions of crypto puzzles are computationally intensive and exhaust CPU cycles for a continuous period of time before their solutions can be computed. Unfortunately, the computation involved in solving these puzzles does not result in any other useful output.

Worse still, crypto puzzles can waste human time, which could have a much higher value than CPU cycles. For example, if a website limits users' accesses to webpages by giving out crypto puzzles that require 10s to solve per webpage download, then users will experience a delay of 10s for every page they see. To help cease this problem, some Proof-of-Work puzzles are designed so that they can be pre-solved; by pre-fetching these puzzles and solving them before actual service requests, the latency incurred can be hidden from the users.

**Non-Interoperability** If a machine has two or more crypto puzzles to solve, all of which require a common resources, then the machine can only solve them one at a time. For example, if a mail server decides that a honest client machine will only send at most one email per minute on average and thus gives out puzzles that require one minute to solve, and a website thinks a honest client machine will only get at most one webpage per minute on average and thus also gives out puzzles that require one minute to solve, then a honest client machine won't be able to *both* send 1 email and visit 1 page per minute! By similar arguments, users can only participate in one Internet-based contest at a time.

## 1.3 Trusted Computing

The term "trusted computing" has come to denote work in the spirit of the *Trusted Computing Group* (TCG) consortium [29]: increasing the security of

standard commodity platforms by adding a small amount of physical hardware and careful integration of software support for it.

In the TCG approach, this hardware takes the form of a *trusted platform module* (TPM) [28], which is an inexpensive chip on the motherboard that participates in the boot process. The TPM maintains a set of *platform configuration registers* (PCRs) that indicate the hardware and software configuration of the platform, and provides services both to release secrets to the platform only if the PCRs show the right values (“unsealing” or “unwrapping”), as well as to prove to remote parties what the current platform configuration is (“attestation”).

This notion of trusted computing thus embodies two security design principles. One is the modern notion of cost-benefit tradeoff: the goal is to improve security without spending too much money. Consequently, one adds a small chip, rather than armoring the entire machine. Another is the classical notion of minimizing the *trusted computing base* (TCB). Although the TPM’s promises of trusted computing initially rest on the assumption that the adversary compromises neither the TPM nor the BIOS, the reality is murkier: attacks on the OS can still subvert protection, and low-cost and highly-effective physical attacks have begun to emerge (e.g., [20]).

## 1.4 Our Contributions

We present an alternative vision for trusted computing: using *Trusted Puzzle Solvers* to construct crypto puzzles with many desirable properties missing from existing constructions. Our solution is secure and efficient, and yet only requires each client to be equipped with minimal tamper-resistant hardware.

**Paper Organization** We explain in Section 2 the details of Trusted Puzzle Solvers, the vital piece of trusted hardware that enables our constructions of crypto puzzles, which are presented in Section 3 and Section 4. We discuss the implications of our design in Section 5 before we conclude the paper in Section 6.

## 2 Trusted Puzzle Solvers

In our design of crypto puzzles to be presented in the next two sections, all client machines are equipped with a hardware module that we call the *Trusted Puzzle Solver*, or *TPS*. We make the assumption that all the functionality provided by these modules are correct and secure, even when subject to certain hardware attacks, such as probing, launched by adversaries in physical proximity. In other words, these modules are the *Trusted Computing Base (TCB)* of our constructions. It is therefore important to minimize their size, in terms of physical volume, circuitry complexity, codebase, and etc, so that we can manufacture them at low costs and yet with high assurance of them meeting our trust assumptions.

Every TPS has a distinct asymmetric key pair (a private key  $sk$  and a public key  $pk$ ),<sup>1</sup> generated and installed by its vendor during manufacturing. The private key  $sk$  resides in, and never leaves, the tamper-proof storage of the TPSs. The public key  $pk$  is certified by one or more Certification Authorities (CAs), such as the TPS vendors, which all servers recognize and trust. Client machines know the public key and the associated certificate  $cert$  of the TPS they are equipped with. Servers know the public keys of the CAs for certificate signing and thus can verify the correctness of the public keys of the TPS modules.

TPSs also contain several other components within their tamper-resistance boundaries. One such component is a clock. The clocks in the TPSs need not be synchronized to a global clock and may be reset at power-on, as long as they all are ticking at the same and reasonably precise frequency. Other components include several registers for storing key materials and internal states, some simple logic for arithmetics and control, and a cryptographically secure random number generator (RNG). Finally, as we will explain in detail in the next section, TPSs further contain the necessary circuit to perform cryptographic operations such as computing HMACs, digital signature signing and public-key decryption.

### 3 Our Proposed Proof-of-Work Puzzles

Recall that the ultimate goal of having clients solve Proof-of-Work puzzles is to rate-limit their service accesses. While existing constructions of Proof-of-Work puzzles achieve this goal by imposing a computational cost on each access, our design relies on the TPS of the client to do the policing: only TPSs know how to solve the puzzles, and TPSs will only solve puzzles up to a certain rate.

More concretely, a puzzle consists of a nonce and a fee. The nonce prevents the clients from replaying puzzle solutions. The fee is a parameter that *precisely* specifies the time it takes to solve the puzzle. A valid solution to a puzzle is a signature signed by a TPS on the puzzle. Thus only TPSs can solve the puzzles.

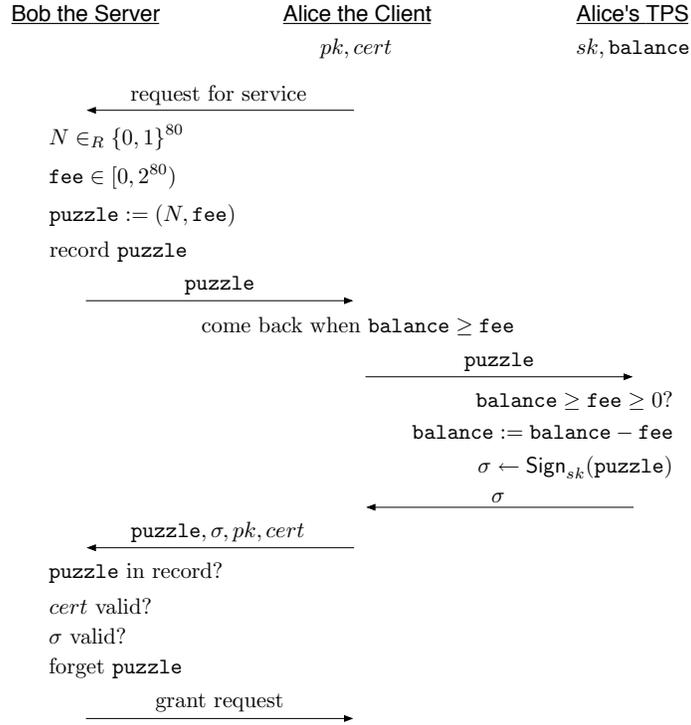
It suffices to make sure the TPS modules don't solve the puzzles too quickly. Each TPS has a register named `balance`, the value of which is incremented periodically. In our construction, it is incremented by 1 every millisecond. TPSs leave their factories with `balance` initialized to zero. The value stored in `balance` may be persistent across up-times or set to zero upon reset. Given a puzzle with a fee, a TPS solves it only if the current balance is no less than the fee, in which case the TPS also deducts the fee from the balance.

#### 3.1 Protocol Description

We now describe our proposed protocol executed between a server and a client, during which the server would like the client to solve a Proof-of-Work puzzle before granting a service request made by the client. We name the client Alice and the server Bob.

We have also included a diagrammatic description of the protocol in Figure 1.

<sup>1</sup> As will become clear, the key pair is for digital signature in our Proof-of-Work puzzles and for public-key encryption in our Time-Lock puzzles.



**Fig. 1.** Diagrammatic Protocol Description of Our Proof-of-Work Puzzles

1. Alice  $\rightarrow$  Bob :  $\langle \mathbf{request} \rangle$   
Alice requests Bob for his service.
2. Bob  $\rightarrow$  Alice :  $\langle \mathbf{puzzle} \rangle$   
Bob challenges Alice to solve  $\mathbf{puzzle} = (N, \mathbf{fee})$ , where  $N$  is a 80-bit random nonce and  $\mathbf{fee}$  is a 80-bit non-negative integer. The  $\mathbf{fee}$  may be a constant or a function of the current load of the server, and/or even the identity of the client. Bob records  $\mathbf{puzzle}$ .
3. Alice examines the  $\mathbf{fee}$  in the  $\mathbf{puzzle}$ . If her TPS doesn't have enough  $\mathbf{balance}$  to pay for the  $\mathbf{fee}$  at the moment, she comes back later when  $\mathbf{balance}$  has become sufficient. In the meantime, she can do something meaningful instead of busy-waiting or stalling. Alice knows when to come back by keeping track of the time and her TPS usage. Alternatively, if Alice thinks that the fee is too expensive, she may refrain from accessing Bob's service by terminating the protocol.
4. Alice  $\rightarrow$  TPS :  $\langle \mathbf{puzzle} \rangle$   
Alice relays  $\mathbf{puzzle}$  to her TPS.
5. TPS  $\rightarrow$  Alice :  $\langle \sigma \rangle$   
Alice's TPS aborts if the current  $\mathbf{balance}$  is less than  $\mathbf{fee}$ . Otherwise, the

TPS deducts `fee` from `balance`, signs `puzzle` with its private key `sk`, resulting in a signature  $\sigma = \text{Sign}_{sk}(\text{puzzle})$ . The TPS returns  $\sigma$  to Alice.

6. Alice  $\rightarrow$  Bob :  $\langle \text{soln} \rangle$   
Alice answers Bob’s challenge with  $\text{soln} = (\text{puzzle}, \sigma, pk, cert)$ .
7. Alice’s  $\text{soln} = (\text{puzzle}, \sigma, pk, cert)$  is valid *if and only if*  $cert$  is a valid certificate on the  $pk$ ,  $\sigma$  is a valid signature on `puzzle` under  $pk$ , and `puzzle` appears in Bob’s record. If  $\text{soln}$  is valid, Bob grants Alice’s service request and removes `puzzle` from his record. He declines the request otherwise.

### 3.2 Analysis

**Properties** Our construction has *none* of the drawbacks we discussed in Section 1.2. First, the time it takes to solve a puzzle is as precise as the clock in the TPS, and thus a PDA can solve as many puzzles as a resourceful desktop PC. Also, there is no need to waste any resources such as CPU cycles on the client machine, or human time waiting for puzzles to be solved as long as there is enough balance in the TPS.

By having the TPS keep a separate balance for each application, our construction allows multiple applications to solve their own puzzles in parallel. The cost per application is only one extra register (for storing the balance). Notice that these extra registers need not be tamper-proof; one can store the balance values in insecure memory outside the TPS by using techniques similar to “sealing” in TCG’s TPMs. Thus, our TPS construction allows interoperability among any number of applications requiring Proof-of-Work puzzles and yet requires only a constant amount of trusted storage.

**Parameters** We have chosen RSASSA-PKCS1-v1.5 [19, §8.1] using SHA-1 as the digital signature scheme used by the TPS because all TPMs that are compliant to the TCG’s TPM v1.2 specification [28] must support it. Nonces are 80-bit and hence picking them uniformly at random prevents nonce reuse. We allow the fee of a puzzle to be any 80-bit non-negative integer. A puzzle is thus only 20 bytes in size. The tamper-resistant register `balance` is 80-bit in size, which will never overflow in practice at the rate of incrementing by 1 per millisecond. In fact, one might want to set a much smaller upper bound for it so as to limit the module’s ability to presolve puzzles.

**Efficiency** The client has nothing to do except relaying a few messages at the right time; the TPS signs one signature on the puzzle. Like any other Proof-of-Work puzzle schemes, there are 4 rounds of communication between the server and the client. Generating a puzzle is efficient, as it involves only picking a random nonce and deciding on the fee of the puzzle. Verifying the solution of a puzzle involves two digital signature verification. For each service request pending for a puzzle solution, the server needs to remember a 20-byte puzzle only.

**Security** A client in possession of a TPS for  $t$  milliseconds has a balance of at most  $t$  at any time instant during the period of the possession. Assume the contrary that the construction is insecure, then the client has been able to correctly solve puzzles such that the sum of their fees exceeds  $t$  during the same

time period. Hence there exists at least one correctly solved puzzle that was not solved by the TPS. Since all puzzles are distinct (due to the nonces in them), the solution to the one puzzle that was not solved by the TPS contains a forged signature, which contradicts to the unforgeability of the digital signature scheme.

## 4 Our Proposed Time-Lock Puzzles

The traditional means of ensuring a puzzle to be solvable only after a predetermined time by requiring the client to go through some tedious operation suffers from all the drawbacks we listed in Section 1.2. Our solution relies on the TPSs present in the client machines as trusted time servers: the TPSs make sure that sufficient time has elapsed before they help the clients solve the puzzles.

Specifically, when a client machine receives a puzzle from the server, it relays the puzzle to its TPS. In a naive solution, the TPS would wait for enough time and then return the solution to the client machine. However, this is undesirable since either the TPS is incapable of concurrently solving concurrent but independent time-lock puzzles, or the TPS must keep in its tamper-resistant memory states of size linear to the number of puzzles allowed to be solved at a time. Rather, in our design, the TPS time-stamps the puzzle, thus witnessing that the client machine has obtained the puzzle at a particular time. When the client machine later comes back with the time-stamped puzzle after sufficient time has elapsed, the TPS will solve the puzzle for the client machine.

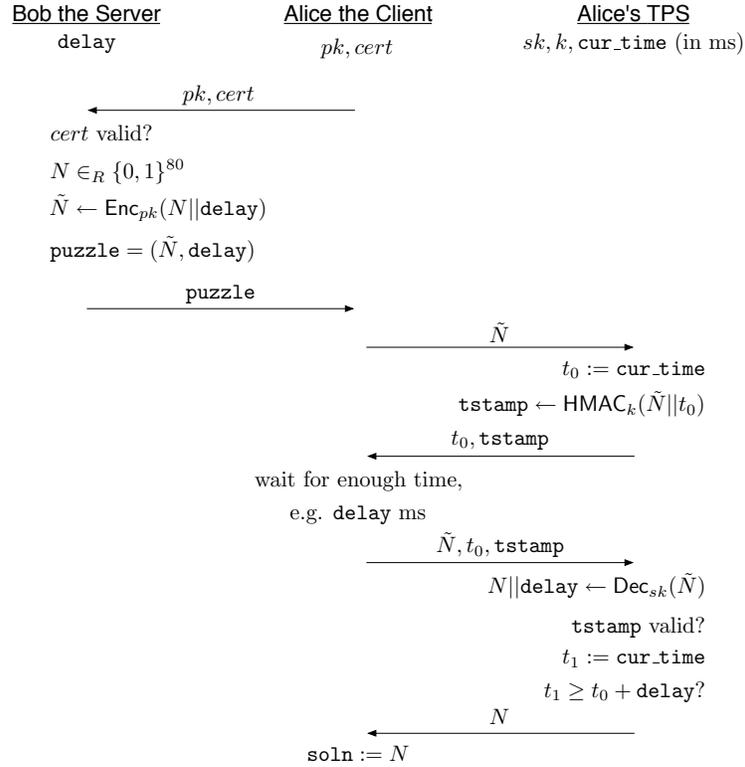
We highlight that although our solution requires each machine to have a TPS, we do *not* rely on the uniformity of the computational resources these TPSs possess. In fact, we envision that as trusted hardware modules, be them TPSs or TCG’s TPMs, become more and more prevalent, vendors will manufacture these modules with different processing power and capability, much like any other components in custody PCs we see nowadays. Our design is secure so long as the TPS hardware satisfies certain basic requirements such as possessing a trusted clock, some trusted circuitry and a few trusted storage.

### 4.1 Protocol Description

We now describe our proposed protocol executed between a server and a client when the client requests a Time-Lock puzzle from the server. Again we name the client Alice and the server Bob.

Figure 2 shows a diagrammatic description of the protocol.

1. Alice  $\rightarrow$  Bob :  $\langle pk, cert \rangle$   
Alice requests Bob for a Time-Lock puzzle by sending him her TPS’s public key  $pk$  and its certificate  $cert$ .
2. Bob  $\rightarrow$  Alice :  $\langle \mathbf{puzzle} \rangle$   
Bob aborts if  $cert$  is not a valid certificate for  $pk$ . Otherwise, he returns Alice with  $\mathbf{puzzle} = (\tilde{N}, \mathbf{delay})$ , where  $\mathbf{delay}$  is a 80-bit non-negative integer that denotes the time (in ms) necessary for solving the puzzle and  $\tilde{N}$  is the



**Fig. 2.** Diagrammatic Protocol Description of Our Time-Lock Puzzles

encryption of a 80-bit random nonce  $N$  concatenated with **delay** under the TPS's public key  $pk$ , i.e.  $\tilde{N} = \text{Enc}_{pk}(N||\text{delay})$ .

3. Alice  $\rightarrow$  TPS :  $\langle \tilde{N} \rangle$   
 Alice relays the  $\tilde{N}$  to her TPS.
4. TPS  $\rightarrow$  Alice :  $\langle t_0, tstamp \rangle$   
 Let the current time be  $t_0$ . The TPS returns  $t_0$  and **tstamp** to Alice, where **tstamp** is the time-stamp on  $\tilde{N}$  at time  $t_0$  under the TPS's secret HMAC key  $k$ , i.e.  $tstamp = \text{HMAC}_k(\tilde{N}||t_0)$ .
5. Alice comes back after sufficient time, i.e. after **delay** ms or more. Again, she is free to do anything in the meantime, rather than busy-waiting or stalling.
6. Alice  $\rightarrow$  TPS :  $\langle \tilde{N}, t_0, tstamp \rangle$   
 Alice sends to her TPS  $\tilde{N}$ ,  $t_0$  and **tstamp**.
7. TPS  $\rightarrow$  Alice :  $\langle N \rangle$   
 Given  $\tilde{N}$ ,  $t_0$  and **tstamp**, Alice's TPS proceeds as follows. It first decrypts  $\tilde{N}$  into  $N||\text{delay}$  using its private key  $sk$ , i.e.  $N||\text{delay} = \text{Dec}_{sk}(\tilde{N})$ . If **tstamp** is valid, i.e.  $tstamp = \text{HMAC}_k(\tilde{N}||t_0)$ , and  $t_1 \geq t_0 + \text{delay}$ , where  $t_1$  is the current time, the TPS returns  $N$  to Alice. The TPS aborts otherwise.
8. The solution to the puzzle is **soln** =  $N$ .

## 4.2 Analysis

**Properties** Our proposed Time-Lock puzzles have all properties we have desired. The puzzles have a solving time as precise as the clock in the TPSs. They are environmental friendly because virtually no resources is wasted by the client in solving them. Finally, one client machine can solve multiple time-lock puzzles concurrently, without any slowdown in solving any of them.

**Parameters** We have picked RSA-ES-OAEP [19, §7.1] using SHA-1 as the asymmetric encryption and HMAC-SHA-1 as the message authentication scheme. Again the support of these functions is required by TPM specification v1.2 [28]. `cur_time` is a 80-bit register that stores the current time in millisecond (relative to, e.g., the time the TPS was last reset) and will never overflow in practice.

**Efficiency** To create a puzzle, Bob needs to do one asymmetric encryption of a one-block plaintext. The TPS has to do two HMACs and one asymmetric decryption of a one-block ciphertext. Alice does not need to do any computation.

Neither the client nor the TPS has to stall when handling a Time-Lock puzzle and thus practically any number of Time-Lock puzzles can be solved concurrently. Also, the TPSs needs not keep any state for each pending puzzle.

**Security** A secure time-lock puzzle cannot be solved earlier than the specified time. We argue in the following that this requirement holds in our construction. Assuming the contrary that a client machine can solve a puzzle without having waited for the specified delay after obtaining the puzzle, then the client must have successfully decrypted the ciphertext in the puzzle before the predetermined time (i.e., the time the machine received the puzzle plus the delay specified in the puzzle). The TPS could not have decrypted the ciphertext as it would imply a forgery of a time-stamp on the puzzle with a time earlier than the time the machine obtained the puzzle, contradicting to the security of the HMAC. As a result, the fact that the client was able to decrypt the ciphertext violates the security guarantee of the encryption, which leads to a contradiction.

It is also crucial for time-lock puzzles to be solvable at the specified time (rather than some time much later). For example, this property is necessary to guarantee fairness in applications such as sealed-bid auctions and Internet-based contests. It is easy to see that our construction of time-lock puzzles enjoys this property. In fact, our construction enjoys it at a stricter sense—even when some client machines are solving more than one time-lock puzzle at the same time.

## 4.3 Realizing Timed-Release Encryption

How to use the Time-Lock puzzles we just proposed is application-dependent. Here, we give an example of using our puzzles to realize Timed-Release Encryption, for applications such as Internet-based contests.

We will need in addition a secure symmetric encryption scheme and a secure cryptographic hash function. Let  $\ell$  denote the length of the symmetric key used in the encryption scheme,  $\mathcal{E}$  and  $\mathcal{D}$  denote the encryption and decryption algorithms respectively, and  $H : \{0, 1\}^* \rightarrow \{0, 1\}^\ell$  denote the cryptographic hash function.

We can implement the symmetric encryption scheme using AES operating under Counter Mode [14] with 128-bit key, i.e.  $\ell = 128$ . We can implement  $H$  using SHA-1 with proper encoding.

To perform timed-release encryption, i.e., to encrypt a plaintext  $P$  so that the resulting ciphertext can't be decrypted by a client Alice until a release-time predetermined by a server Bob, Bob does the following:

1. Create a Time-Lock puzzle for Alice with delay equal to the difference between the release-time and the current time in millisecond, according to the protocol we described in Section 4.1.
2. Generate a symmetric encryption key  $s$  by hashing the nonce  $N$  in the puzzle, i.e.  $s \leftarrow H(N)$ .
3. Encrypt the plaintext  $P$  under the symmetric encryption key  $s$ , resulting in a ciphertext  $C$ , i.e.  $C = \mathcal{E}_s(P)$ .
4. Return  $C$  to Alice along with the Time-Lock puzzle.

Consequently, by the time Alice has solved the puzzle, she can reconstruct the symmetric encryption key  $s$ , decrypt the ciphertext  $C$ , and recover the original plaintext  $P$ , i.e.  $P = \mathcal{D}_s(C)$ .

**Security** To see why such a Timed-Release Encryption is secure given that the underlying Time-Lock puzzle is secure, consider the following arguments. Assume the contrary that the above Timed-Release Encryption is insecure, then the adversary can learn information about the plaintext from the ciphertext prior to the predetermined release-time. Since the underlying Time-Lock puzzle is secure, the puzzle leaks no information about the solution to the puzzle to any computationally bounded adversary at the time when the adversary learned the information. Now the security of the cryptographic hash function implies that the symmetric key was picked uniformly at random in the computational sense. Therefore, the adversary's ability to learn information about the plaintext from the ciphertext contradicts the security of the symmetric encryption scheme.

## 5 Discussion

**Feasibility** The use of trusted hardware has been conceived by some as waving a magic wand: like invoking magic, it's just unrealistic. We have different opinions. All cryptographic security measures require, at the very least, the secure execution of the cryptographic algorithms and a secure storage of the keys; it is assumed, implicitly or explicitly, that some type of computing engine and storage lies beyond the reach of the adversary. Using trusted hardware simply makes this assumption explicit; designing it requires explicitly thinking about the types of physical attacks an adversary might mount. Deploying it in the real world requires considering tradeoffs between strength of the tamper protections, robustness, cost, and computational power. At one extreme, a powerful piece of trusted hardware such as an IBM 4758 or 4764 secure coprocessor can securely host a (small) application in its entirety but is very expensive; at the other end,

a resource-constrained piece of trusted hardware such as a TPM chip is inexpensive but does little beyond several primitive cryptographic operations. (We note that both these devices exist in the real world—and that TPMs now are ubiquitous in nearly all new desktop and laptop platforms.)

The real challenge when designing a solution to a security problem is how to cost-effectively provide sufficient security with confidence through intelligently minimizing the TCB. Thus, one of our goals when designing TPSs was to make their wide deployment feasible by minimizing the functionality requirements on them. We aimed for something the size of a TPM. In fact, most of the functionality needed by a TPS, such as HMAC-SHA1 and RSA operations, is already present in TCG’s TPMs in their current specification, with the exception of a trusted clock. One could therefore expect that the cost of a TPS is comparable to, if not less than, that of a TPM (assuming that TPS manufacturing enjoys the same economy of scale that TPM manufacturing does).

**TPS Compromises** Given that it is not impossible to circumvent the tamper-resistance of TPSs, it is worth looking at the security implications when tampering happens. In both of our puzzle constructions, if an adversary can read the registers, he learns the private signing/decryption key, and can then solve puzzles at any rate. If the adversary can write to the registers or make the clock tick faster, then he can increase the balance in case of Proof-of-Work puzzles, or fast-forward the current time in case of Time-Lock puzzles. In both cases, the adversary can solve puzzles faster than he should have been able to.

As a defensive mechanism, servers should audit TPS activities, become suspicious if they see some TPSs solving puzzles too quickly and eventually declare those TPSs to have been compromised. Servers should revoke all compromised TPSs by, e.g., using Certificate Revocation-Lists (CRLs) in X.509 [1], which is a revocation mechanism currently also used by TCG’s TPMs.

**Bot-nets** Nowadays spammers seldom use their own machines for spamming. Rather, they take over machines on the network through the use of, e.g. worms and malware, and “steal” resources, e.g., computational power, electrical power, email addresses and IP addresses, from these “zombies” machines to send unsolicited bulk emails. Conventional Proof-of-Work puzzles become ineffective when spammers can summon their can zombies to solve puzzles for them. The problem here is that these puzzles are solver-anonymous: anyone can solve a puzzle and the solution contains no trail of who produced it. Our proposed Proof-of-Work puzzles are more resilient to bot-nets than conventional puzzles because the solution to a puzzle contains the identity of the solver’s TPS. This provides some clues to the server whether the solving of the puzzles has been outsourced—and can potentially be addressed by revocation.

**User Anonymity** As discussed, being able to identify the TPSs helps revoke compromised TPSs and resist bot-nets. However, this implies that our proposed puzzles do not protect the anonymity of the user. In the case of Proof-of-Work puzzles, the server knows which TPS has solved the puzzle by looking at the

digital signature; in the case of Time-Lock puzzles, the client must reveal the public key of her TPS to the server when obtaining a puzzle.

Our proposed Proof-of-Work puzzles can be modified to provide user anonymity as follows. TPSs sign a group signature [5] instead of a digital signature to hide their identity among the set of all TPSs. Since TPM v1.2 implements *Direct Anonymous Attestation (DAA)* [8], v1.2 TPMs already have the necessary circuitry to implement TPSs with user anonymity. We point out that the use of group signatures makes puzzle solution verification more computationally intensive. Augmenting user anonymity to our proposed Time-Lock puzzles in an efficient way seems to be a lot more complicated. We thus leave it as future work.

**Isn't the TPS PKI Enough?** Since each TPS possesses a certified key pair, a wide deployment of TPS modules implies a global Public-Key Infrastructure (PKI) as well. One might think that the mere existence of a global PKI would suffice to mitigate spam and DoS attacks and thus using TPS to do the same would be redundant. We believe this is not the case, for several reasons.

For PKI to be effective in deterring attacks, the certificates need to meaningfully bind a client to a key pair. Large-scale PKIs that do this have proven an elusive and expensive proposition. However, the TPS PKI merely needs to assert a key pair belongs a genuine TPS; as with the TPM PKI, already in existence, it omits the expensive part.

Furthermore, a PKI alone won't solve the problem. Even if we had a global PKI, using it to fight spam and DoS attacks would still require us to ensure the integrity of the client software and to isolate malware from the private keys; these tasks rely on the security of the operating system and the hardware. This almost dictates us to put the entire computing platform into the TCB, which is not only costly but also infeasible. In our solution, the TCB only consists of a small and simple hardware module.

By similar arguments, requiring authentication at the client side using, e.g. TLS, during web browsing does not mitigate DoS attacks. In fact, the extra communications and cryptographic computation required by TLS might actually open up a new DoS attack surface.

**Puzzle Pricing** Regardless of whether real currencies, traditional computation-based crypto puzzles or our TPS-based puzzles are used, pricing the service accesses right is the key to the effective mitigation of spam and DoS attacks without adversely impacting the honest users. For instance, a web server may want to hand out puzzles that are more difficult to solve as its load increases. Similarly, some have suggested that the price for emailing to a mailing list should be function of how big and commercial the list is.

Our TPS-based puzzles facilitate correct pricing better than existing puzzles for a number of reasons. First, the fee of a puzzle can be set at a very fine-grained level. Second, the effort to generate a puzzle and verify its solution is independent of its fee. Third, the time it takes to solve a puzzle is very precise.

## 6 Conclusions

We have proposed the use of trusted computing in designing two types of crypto puzzles, namely Proof-of-Work puzzles and Time-lock puzzles. In particular, we have presented how to construct these puzzles by assuming that each client machine is equipped a *Trusted Puzzle Solver*, or *TPS*, which is a small tamper-resistant hardware module. Our proposed crypto puzzles are the first that achieve all the aforementioned desirable properties simultaneously, and can thus be used to effectively combat spam and DoS attacks. Our analysis has shown that our designs are secure and efficient.

TPSs are cost-effective and trustworthy because of their simplicity. Almost all the necessary functionality is already present in TCG's TPM architecture today. These factors make it likely for TPSs to be widely deployed.

## Acknowledgment

We would like to thank Jianying Zhou and the anonymous reviewers for their helpful comments.

## References

1. C. Adams and S. Farrell. Internet X.509 Public Key Infrastructure Certificate Management Protocols. Internet Engineering Task Force: RFC 2510, 1999.
2. T. Aura, P. Nikander, and J. Leiwo. DOS-Resistant Authentication with Client Puzzles. In *Security Protocols Workshop*, volume 2133 of *LNCS*, pages 170–177. Springer, 2000.
3. A. Back. Hashcash, 1997. <http://hashcash.org>.
4. M. Bellare and S. Goldwasser. Encapsulated Key Escrow. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1996.
5. M. Bellare, H. Shi, and C. Zhang. Foundations of Group Signatures: The Case of Dynamic Groups. In *CT-RSA*, volume 3376 of *LNCS*, pages 136–153. Springer, 2005.
6. D. Boneh and M. Naor. Timed Commitments. In *CRYPTO*, volume 1880 of *LNCS*, pages 236–254. Springer, 2000.
7. N. Borisov. Computational Puzzles as Sybil Defenses. In *Peer-to-Peer Computing*, pages 171–176. IEEE Computer Society, 2006.
8. E. F. Brickell, J. Camenisch, and L. Chen. Direct Anonymous Attestation. In *ACM Conference on Computer and Communications Security*, pages 132–145. ACM, 2004.
9. J. Cathalo, B. Libert, and J.-J. Quisquater. Efficient and Non-interactive Timed-Release Encryption. In *ICICS*, volume 3783 of *LNCS*, pages 291–303. Springer, 2005.
10. K. Chalkias and G. Stephanides. Timed Release Cryptography from Bilinear Pairings Using Hash Chains. In *Communications and Multimedia Security*, volume 4237 of *LNCS*, pages 130–140. Springer, 2006.
11. A. C.-F. Chan and I. F. Blake. Scalable, Server-Passive, User-Anonymous Timed Release Cryptography. In *ICDCS*, pages 504–513. IEEE Computer Society, 2005.

12. D. Dean and A. Stubblefield. Using Client Puzzles to Protect TLS. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 1–1, Berkeley, CA, USA, 2001. USENIX Association.
13. C. Dwork and M. Naor. Pricing via Processing or Combatting Junk Mail. In *CRYPTO*, volume 740 of *LNCS*, pages 139–147. Springer, 1992.
14. M. Dworkin. Recommendation for Block Cipher Modes of Operations—Methods and Techniques. Technical report, National Institute of Standards and Technology (NIST), Dec 2001. <http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf>.
15. M. K. Franklin and D. Malkhi. Auditable Metering with Lightweight Security. In *Financial Cryptography*, volume 1318 of *LNCS*, pages 151–160. Springer, 1997.
16. J. A. Garay and M. Jakobsson. Timed Release of Standard Digital Signatures. In *Financial Cryptography*, volume 2357 of *LNCS*, pages 168–182. Springer, 2002.
17. M. Jakobsson and A. Juels. Proofs of Work and Bread Pudding Protocols. In *CMS '99: Proceedings of the IFIP TC6/TC11 Joint Working Conference on Secure Information Networks*, pages 258–272, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.
18. A. Juels and J. G. Brainard. Client Puzzles: A Cryptographic Countermeasure Against Connection Depletion Attacks. In *NDSS*. The Internet Society, 1999.
19. B. Kaliski and J. Staddon. PKCS #1: RSA Cryptography Specifications Version 2.0, 1998.
20. B. Kauer. OSLO: Improving the Security of Trusted Computing. In *USENIX Security Symposium*, pages 229–237. USENIX, 2007.
21. D. Mankins, R. Krishnan, C. Boyd, J. Zao, and M. Frenz. Mitigating Distributed Denial of Service Attacks with Dynamic Resource Pricing. In *ACSAC '01: Proceedings of the 17th Annual Computer Security Applications Conference*, page 411, Washington, DC, USA, 2001. IEEE Computer Society.
22. T. May. Time-release Crypto. manuscript, Feb 1993.
23. R. C. Merkle. Secure Communications Over Insecure Channels. *Commun. ACM*, 21(4):294–299, 1978.
24. R. L. Rivest. Description of the LCS35 Time Capsule Crypto-Puzzle, April 1999. <http://www.lcs.mit.edu/about/tcapintro041299>.
25. R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock Puzzles and Timed-release Crypto. Manuscript. Available at <http://theory.lcs.mit.edu/~rivest/RivestShamirWagner-timelock.ps>.
26. R. Roman, J. Zhou, and J. Lopez. Protection Against Spam Using Pre-Challenges. In *SEC*, pages 281–294. Springer, 2005.
27. A. Serjantov and S. Lewis. Puzzles in P2P systems. In *8th Cabernet Radicals Workshop*, Oct 2003.
28. TPM Work Group. TCG TPM Specification Version 1.2 Revision 103. Technical report, Trusted Computing Group, 2007.
29. Trusted Computing Group. TCG Specification Architecture Overview Revision 1.4. Technical report, Trusted Computing Group, 2007.
30. L. von Ahn, M. Blum, N. J. Hopper, and J. Langford. CAPTCHA: Using Hard AI Problems for Security. In *EUROCRYPT*, volume 2656 of *Lecture Notes in Computer Science*, pages 294–311. Springer, 2003.
31. B. Waters, A. Juels, J. A. Halderman, and E. W. Felten. New Client Puzzle Outsourcing Techniques for DoS Resistance. In *ACM Conference on Computer and Communications Security*, pages 246–256. ACM, 2004.